Manual exec.: BFS, DFS, Ford's Alg., Djikstra (+reverse), A*, Prim's Alg., DAG topo. sort by predecessor count, DAG topo. sort by DFS, Floyd-Warshall, Ford-Fulkerson, Kruskal, Vertex cover

BFS - breadth-first search

- iterative w/ queue (FIFO)
- check if 7 paths
- same complexity
- add start node to queue →
→ pull a node → process if not seen →
→ add each unseen adjacent node to queue

DFS - depth-first search

- on a stack (LIFO) can use recursion
- complete search, finding all paths, ...

Time: $O(|V|+|E|)$; Space: $O(|V|)$

add start node
↑ Pull node from stack → if not seen:
add to seen & process → add its children on the STACK (if unseen)

Ford's Algorithm (Bellman-Ford)

- shortest path from a source vertex to all the other vertices
- V-1 iterations (V is the number of vertices)

at start:

S	V_1	V_2	- - - -
0	∞	∞	

starting vertex

1st iteration

0	10	∞	∞	∞	8
(S)	A	B	C	D	E

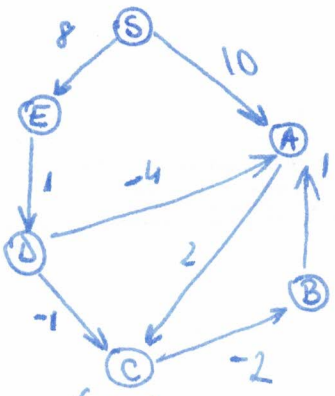
curr. node

0	10	∞	12	∞	8
S	(A)	B	C	D	E

0	10	~~10~~ ∞	12	∞	8
S	A	B	(C)	D	E

0	10	10	12	9	8
S	A	B	C	D	(E)

- we then take node (A) and see we can reach to node (C)
- we take (B), we don't know how to reach it so we skip
- we can get to (B) from (C)
- we skip (D)
- we get to (D) from (E)



(final: S A B C D E
0 5 5 7 9 8)

... iterate a total of V-1 times & update only when we get a shorter path

ex: Iteration 2, when reaching (D), the paths to (A) & (C) change

0	~~10~~ 5	10	~~12~~ 8	9	8
S	A	B	C	(D)	E

!! if no values change during an iteration, the execution can be stopped

Djikstra's Algorithm - shortest path between ~~two vertices~~ the starting vertex to all other vertices

(outbounds)

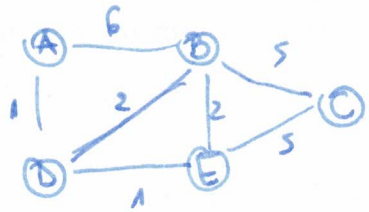
- take the neighbours of the starting vertex & compute the distances; update the values if they are smaller than the ones in the table
- mark the current node as 'visited'
- visit the vertex with the smallest known distance from the previous one (unvisited)
- re-calculate the distances from the starting vertex & update if necessary

starting vertex

S	0
V_1	∞
...	
V_m	∞

initial table

ex: A - starting vertex



① A → D = 1
A → B = 6

A	0
B	6
C	∞
D	1
E	∞

A → visited

② we take vertex D

→ B: 1+2=3 < 6 - UPDATE
→ E: 1+1=2 < ∞ - UPDATE

A	0
B	3
C	∞
D	1
E	2

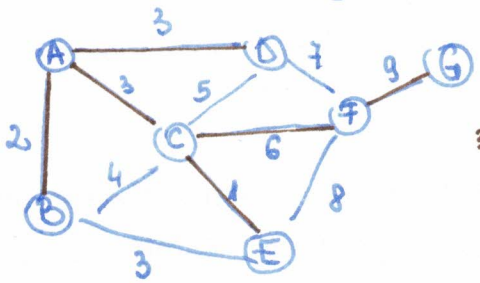
D ⟷ visited
we take E ③

A* Algorithm - path finding & graph traversals
- an extended (faster) Djikstra that also depends on a heuristic h (take the vertices based on the cost + h)

Prim's Algorithm - for finding a Minimum Spanning tree (MST)

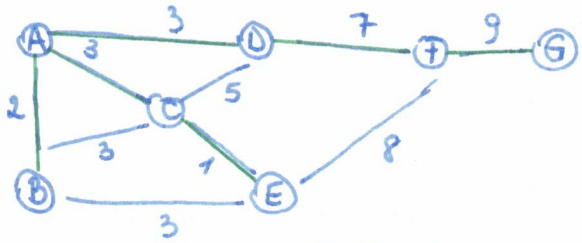
MST - min. weight connected graph with no cycles

→ add starting vertex to 'visited'
→ choose the smallest edge that connects to an unvisited node (& mark that node as 'visited')
→ take the smallest edge from the 'visited' nodes



1) A - start, A - visited
2) Take [AB], B - visited
3) we can take [AD], [AC], [BE] (doesn't matter)
...

Kruskal's Algorithm ⇒ results in a MST



→ pick the smallest edge : [EC]
→ look for the smallest edge that doesn't create a cycle : [AB]
→ choose any from [AD], [AC], [BE] (for ex: [AD])
→ pick [AC] next
→ picking them [BE] would result in a cycle
(nodes are alr. in the tree)
→ next is [DT] & [TG]

DAG - Directed Acyclic Graph (dir. graph w/ no cycles)

Topological sorting using the predecessor counting algorithm
→ we take a vertex w/ no predecessors, we put it on the sorted list & we eliminate it from the graph, and continue the same way
→ ~~vertex w/~~ if we cannot get a vertex w/ no predecessors ⇒ we have a cycle

DFS-Based topological sorting (?)

Floyd-Warshall Algorithm - find shortest ~~path~~ distance between every pair of vertices in a given directed graph

→ create a V x V matrix initially having ∞ (• matrix ← dist)
(dist[i][i] = 0)
→ we put the edges first (dist[u][v] = cost(u,v))
→

```
for k from 1 to V
    for i from 1 to V
        for j from 1 to V
            if dist[i][j] > dist[i][k] + dist[k][j]
                (replace the value)
```